INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1518

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

AUTOMATIC ANALYSIS AND TRANSFORMATION OF FORTRAN PROGRAMS USING A TYPED FUNCTIONAL LANGUAGE

Nicole ROSTAING
Stéphane DALMAS

Septembre 1991

RR-1518

Programme 2
Calcul symbolique, Programmation et Génie logiciel

Analyse et transformation automatiques de programmes Fortran à l'aide d'un langage fonctionnel typé

Nicole Rostaing [1]
rostaing@mirsa.inria.fr

Stéphane Dalmas [2]
dalmas@mirsa.inria.fr

Résumé

Notre objectif est de faire du Calcul Formel avec des objets mathématiques représentés par des programmes Fortran. Notre approche se fonde sur l'utilisation d'un langage fonctionnel polymorphe typé, CAML. Dans cet article, nous expliquons pourquoi nous avons choisi un tel langage et nous décrivons les premiers composants de ce qui deviendra une "boite à outils" pour l'analyse et la transformation de programmes Fortran. Nous présentons aussi deux exemples d'application.

Automatic analysis and transformation of Fortran programs using a typed functional language

abstract

We are interested in symbolically manipulating mathematical objects represented as Fortran programs. Our approach is based on the use of a polymorphic functional language, CAML. In this paper, we explain the advantages of this choice and the construction of the first components of what will be a toolkit for program analysis and transformation. Two examples of applications are presented.

[1]Projet SAFIR: Université de Nice-Sophia Antipolis et INRIA Sophia Antipolis

[2]Projet SAFIR, CNES

implemented. The other has reference to program transformation. It is a large subject, we then just give some ideas of how we can easily use the tools we describe.

This work is done within the SAFIR Computer Algebra group [2], it is part of the thesis work of the first author, directed by André Galligo [3].

2 Program structures to be derived

When a scientist or an engineer writes a Fortran program, he translates a numerical method for solving a mathematical problem or an idea of a numerical simulation into a programming language He keeps in mind the mathematical semantic of his problem and uses the syntax of Fortran to express it.

We have to do the job in the opposite direction. We read a Fortran "text" and try to find a meaning for the "phrases".

From each compilation unit (*program, subroutine, function, or block data*), we extract some information about the variables (where they are declared, their types...), the control structures (to know whether conditional structures or loops are used), the Fortran functions or subroutines called, and so on. Then, we transform each compilation unit after another into a *typed abstract syntax tree*:

- *tree*: it is the recursive structure we give to the text. The root is an operator whose arguments are such trees, called *sub-trees*. The sub-trees may be reduced to *leaves* in the case of variables or constants.
- *abstract syntax*: the structure is very close to the Fortran syntax : the operators express without ambiguities, the meaning of the syntactical forms. In that sense, it is an abstract syntax. The Fortran syntax is concrete.
- *typed*: A type is assigned to each argument of an operator, related to its meaning.

After this processing, all of the Fortran "expressions" are then typed terms. A treatment on a Fortran compilation unit consists of its abstract syntax tree traversals for extracting some information or for tree transformations performing. The types help us to be sure that a given transformation is valid for the object on which it is applied. The typechecking mechanism ensure a certain consistency.

Parsing is sometimes not sufficient to deduce all of the information we need about variables : a semantic analysis is then indispensable. For example, it is impossible to make the difference between function call and array indexing without knowing the nature of the variables. Consequently, all of the features

[2] SAFIR group is a collaboration between INRIA and the University of Nice-Sophia Antipolis

[3] Professor at the University of Nice-Sophia Antipolis

about the current Fortran unit variables are gathered in a symbol table, joined to the tree.

3 A brief presentation of the CAML language

3.1 A typed functional language

The CAML language is a functional language in the ML (*metalanguage*) family. ML languages are based on a version of the λ-calculus theory. They are strongly typed: they use a polymorphic type system allowing type inference. Every typable ML expression possesses a most general type which may be synthetized by the language compiler. The ML dialects are functional languages :

1. their basic component is the notion of *recursive function*
2. functions are first class values : they can be used as arguments and returned by functions.
3. their essential control structure is *function application*.
4. they use *pattern-matching*. Functions are defined by patterns (i.e. induced by the forms of their arguments).

They also allow non functional programming (imperative features, side effects). The user has not to manage the memory : the memory allocation is dynamic, using a garbage collector.

The CAML compiler produces CAM (Categorical Abstract Machine) code which can be expanded into efficient machine code.

CAML is also an interactive language. It is possible to type an expression at the toplevel. This expression is parsed, analysed for type correctness, compiled and executed. The system then answers either with an error message or prints a representation of the value of the expression and its type.

3.2 CAML is adapted to our purpose

An interface with the Yacc parser generator is available in CAML running under the Unix system. This gives to the user the possibility of conveniently associating an abstract syntax tree to a concrete syntax. The user provides a description of the syntax for an object language (Fortran in our case) as a set of grammar rules. CAML statements (actions) are attached to these rules, and the corresponding values are computed when a production is reduced.

We use this mechanism and the possibility offered by the language to define our own data types, to analyse the Fortran concrete syntax, like a compiler, but instead of producing machine code, we just produce the corresponding typed abstract syntax tree.

In the following, we will show more precisely how we fully used the main features of the CAML language :

- pattern-matching: to manipulate the trees representing Fortran programs.
- polymorphism: to write powerful iterators.
- dynamic memory allocation

The prototype we made looks adapted to an industrial application : in a forthcoming section, we will describe some tests we realised. What is more, CAML is portable and it is possible to make standalone executables.

4 Description of the CAML program to analyse a Fortran compilation unit

4.1 From the concrete to the abstract syntax

From the Fortran text to the typed abstract syntax tree and the symbol table, the steps are quite classical:

1. *Scanning*:
   The lexical analyser recognizes Fortran keywords and transforms the input text into a sequence of tokens.
2. *Parsing*:
   A LR(1) bottom-up parser (generated by the Yacc interface) recognizes the syntax and performs the semantic actions attached to the concerned grammar rules. They are responsible for building the abstract syntax tree.
3. *Static semantic analysis*:
   Some information about the variables may not be deduced only from the syntax: whether a variable has been declared or not, with which type ... We find this information during parsing and record it in a symbol table.

4.2 Implementation

4.2.1 About the lexical analysis

We have used an existing lexical analyser written in C.

Certain Fortran features and our aim to write a simple and well structured grammar leaded us to enlarge the role of the lexical analyser in two areas:

- clear separation between the specification and statement parts of the program.
- handling of DO loop termination

A Fortran compilation unit is essentially composed of a block of specifications followed by a block of statements, specifications and statements cannot be mixed (that is what we like to express in the abstract syntax). Unfortunately, Fortran allows a keyword like ENTRY to appear both in the specification part and in the statement part. We identify it when scanning and generate two different keywords : **DECLENTRY** when it is a specification and **STATENTRY** otherwise. When parsing, there is no more ambiguity: at each keyword will be associated a specifical term. It is not an artificial difference, it corresponds to a certain semantic difference.

Another major drawback of the Fortran grammar is the lack of a terminating keyword for the loop construct. Taking this into account at the LR(1) parser level (in the semantic actions) would have dramatically complicated its structure. So we let the scanner do the work, managing loops and generating the missing keyword ENDDO.

4.2.2 The abstract syntax in CAML

From the Fortran concrete syntax, we elaborate an abstract syntax, represented as a set of CAML concrete types.

As we said, a compilation unit is a program, a subroutine, a function or a block data. It is expressed in the CAML language by defining the type **CompilationUnit** with four *type constructors* : **'Program, 'Subroutine, 'Function, 'BlockData**. Such a type, defined by a name, and a set of constructors, is called a *concrete type* by CAML designers. Its complete definition is :

```
type CompilationUnit =
     'Program of Name option * Spec list * Stat list
   | 'Function of
        Name * BasicType * Var list * Spec list * Stat list
   | 'Subroutine of Name * Var list * Spec list * Stat list
   | 'BlockData of Name option * Spec list
```

The second line means that we see a Program as composed of an optional name, specifications (declarations, commons, equivalence...), and statements (the body of the program).
We obtain a typed term whose operator is **'Program** and the arguments are a name of type **Name**, a list of specifications of type **Spec** and a list of statements of type **Stat**.

The two concrete types **Spec** and **Stat** correspond to the two Fortran syntactic classes : the specifications and the statements. Their respective constructors have reference to each Fortran syntactic form.

An example of specification is the declaration of variables. We represent it with the constructor **'Decl** whose argument is a list composed of variables and their types:
'Decl of (Var * Type) list describes that we attach at each variable its type.

Let us give two examples of statements:

- the assignment: **expr1 = expr2**
  by **'Let of Expr * Expr**
- or DO loop by : **'Do of Var * Expr * Expr * Expr option * Stat list**
  meaning that a loop is composed of a variable which will be incremented from an initial value to a final value with a step value (optional when it is one) and a sequence of statements which composes its body.

We can see here that the type **Stat** is recursive (a statement can contain a statement), allowing us to naturally express that loops may be nested or contain conditional branches ...

4.2.3 Symbol table

Our analyser attaches a symbol table to the syntax tree of a compilation unit. This table holds the basic information about the identifiers (variable, subroutine and function names) used in that unit: if they are local, global (and their common block name) or dummy argument and their types.

This table is implemented as a hash table with chaining.

Analysing Fortran declaration is as not easy as it might seem. The exact type of a variable can come from many sources (implicit rules, explicit type declarations, dimension declarations), in different orders, and mixed with other information (external specification, global from a common block ...). The symbol table must be successively updated.

4.2.4 Generating Fortran code

Of course, we need to be able to generate a valid Fortran code from an abstract syntax tree. It is the opposite work of parsing. There is really no problems here because the mapping between our abstract syntax trees and Fortran code (in textual form) is very close. The CAML language provides a dedicated set of tools (printers and pretty-printing grammars) to print the values of a concrete type.

5 Two examples of applications

The typed abstract syntax tree and the associated symbol table being produced, we can use them easily thanks to the possibility of writing functional polymorphic iterators for tree and table processing.

5.1 Programming norms verification

Researchers on weather forecasts problems [4] provide us with a good example of a first application of our tool: they want to enforce programming rules on the identifiers names. This is necessary because their code is large and developped by many people, in different places.

In fact, to ensure a good lisibility and maintenance of the programs they set rules on the first or the two first letters of identifiers with respect to their nature (local, global, parameter, formal parameter) and their type (integer, real, complex,...).

Using our CAML tools, we were able to quickly write a program printing informative messages about non valid variable names.

5.2 Code transformation

The motivation of our work is to perform symbolic manipulations on mathematical objects represented as Fortran programs. We want to apply a transformation on a piece of Fortran code and obtain a new piece of Fortran code. This transformation is a symbolic manipulation on the tree representing the input Fortran code. Code transformation is thus done in two successive steps:

1. Tree transformation
2. Code generation from the transformed abstract syntax tree

Usual computer algebra operates on algebraic expressions. If we want to operate on code, we have to take into account the dynamic structure (control and iterative structures) of the program containing algebraic expressions. So our tree transformation reflects these two aspects: the algebraic processes use classical computer algebra methods (could be done by a computer algebra program) and the other manipulations need some adapted tools.

These tools will fully use the CAML facilities like pattern-matching to recognize the structures and easily replace a sub-tree by another one, and the polymorphism for writing powerful iterators. The typechecker warrants a certain semantic coherence of the tree obtained after substitutions.

For example, to make a transformation which only affects the statements in the body of a loop, we just replace the statements list in the 'Do term by a new list. We can write a polymorphic CAML function parameterized by the transformation we want to apply on the loop body. It could be seen as a generic iterator operating on the tree structure. It is also possible to write iterators which process more complicated cases taking into account other parameters of the loop like the bounds, loop variables,...

[4] O.Talagrand at LMD (dynamic meteorological laboratory at ENS) and Ph.Courtier at CRMD (dynamic meteorological research center)

We may wonder about the mathematical relevance of a given program transformation. It is a difficult problem, but we are just describing a set of tools, some of them could be helpful in determining the mathematical or numerical validity of the transformation.

6 Usability of the tools

We would like to emphasize two aspects of our approach. We can write quite easily and safely tools for program analysis and transformation based on an extensible library of basic components. We are able to deal with "real life examples".

Let us recall the main advantages of using CAML in this context:

- free the user of explicitly managing memory allocation (source of many subtle bugs)
- providing a very natural representation of abstract syntax trees by means of concrete types.
- tree traversals and rewriting are well expressed with the pattern matching facility.
- writing powerful polymorphic iterators for tree traversals and symbol table processings.

We can illustrate our purpose by making a quantitative estimation of our work by a crude method : counting the number of code lines we needed to write. Our grammar, including semantic actions is 740 lines long. To build and manipulate Fortran expressions, 780 lines of CAML code are necessary. Our analyser, including the abstract syntax definition contains only 550 lines of CAML code. It took us about three weeks to write the presented basic components. The first application (programming norms verification) is 90 lines long and took us half a day.

Although this may seem a bit difficult, it is interesting to compare our approach with the more usual one: writing the tool in the C language. What we like to show is that there is no conclusive advantages to write this kind of programs in such a "low level" language as C. The C language does not possess the CAML programming facilities. Its only benefit may be speed which is not our main interest here.

Anyway, to have a practical idea of the performance and the correctness of our analyser written in CAML, we chose as a target the NAG Fortran library. Our benchmark analyses in turn each subroutine and function of this library, producing the corresponding abstract syntax tree and symbol table. The time spent on a DECstation 5000/200 processing these 298500 lines of code in less than 10 minutes (595 seconds).

The achieved rate of 500 lines per second is quite competitive with current compilers written in C. The same experiment using a Fortran 77 compiler (with linking, optimisation and all outputs suppressed) gives a very similar rate (510 lines per second).

This seems to show that functional languages are not as inefficient as it is sometimes claimed and that our tool will be able to treat examples of "industrial" size.

7 Conclusion and perspectives

The work presented is the basic component of what will be a toolkit for Fortran programs analysis and transformation. This toolkit can be used to make some experiments as well as to produce some tools suitable on "real life" problems.

Among the applications that can be planned, two classes may be derived. The first one is related to totally algorithmic code transformations. In such cases, we are sure that our tool can be substituted to the human doing presently the transformation by hands. The second one concerns transformations where the human experience seems indispensable. In such cases, we could provide a set of tools for assisting the programmer. In this perspective, the collaboration with the LMD and CRMD teams (which work on midle-range weather forecasts) will be very fruitful: the problems raising about the production of the adjoint program of a direct code [1] contain these two aspects.

This work provides a new opportunity to apply some symbolic computation tools to numerical computation. The usual approach is to work on algebraic expressions written in a computer algebra system and generates Fortran code. Our approach directly operates on existing large Fortran programs.

Acknowledgements

We would like to thank P. Weis, of the CAML team at INRIA for advice and encouragements. We also are grateful to O.Talagrand and Ph.Courtier for the discussions we had with them.

References

[1] O. Talagrand. The use of Adjoint Equations in Numerical Modelling of the Atmospheric Circulation. In *Proceedings SIAM Symposium on Automatic Differentiation of Algorithms*, To appear 1991.

[2] P. Weis and al. *The CAML Reference Manual.* INRIA, 1990.

[3] M. Mauny. Beyond Dark CAML , Course notes, Ecole GRECO Jeunes chercheurs en Programmation. 1991.

[4] T.M.R.Ellis. *FORTRAN 77 Programming.* Addison Wesley, 1990.

[5] Masao IRI Kuochi KUBOTA. *PADRE2, version1, Usuer's Manual.* Technical Report, Tokyo University, 1990.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[7] J.Srivasan A.Griewank, D.Juedes. *ADOL-C, version1.* Technical Report, Argonne National Laboratory, Illinois USA, 1990.

ISSN 0249 - 6399